Application Analysis Driven ASIP-based System Synthesis for ECG

Erkan Diken, Roel Jordans, Rosilde Corvino, Lech Jozwiak
Faculty of Electrical Engineering
Eindhoven University of Technology
Eindhoven, The Netherlands
Email: {e.diken, r.jordans, r.corvino, l.jozwiak}@tue.nl

***Abstract*—This paper discusses an analysis-driven design of an application specific instruction-set processor (ASIP) for the electrocardiogram (ECG) medical application. It demonstrates how different tools for application analysis and profiling can be exploited to optimize the application software, as well as, to configure and extend the corresponding VLIW ASIP processor. Appropriate analysis tools are used to extract information on the relevant application characteristics, namely execution bottlenecks, code structure, execution frequencies of the arithmetic and logical operations, etc. The information is subsequently used to guide the design optimizations. Finally, the decided optimizations are carried out manually by the designer. The quality metrics expressing performance and energy consumption are used to evaluate both the proposed software and hardware optimizations.**

I. Introduction

Medical devices play a key role in the biomedical field to improve the quality of health care. In particular, devices capable of monitoring the physiological signals generated by the human body, such as the heart beat, are the main source of information for the cardio-related disease diagnosis. Electrocardiogram (ECG) application is used to measure the electrical activity of the heart and to subsequently analyze the measured heart electrical signals for detection of a potential heart disease or malfunction.

ECG can be used to monitor the response of a patient's heart to everyday life activities and can warn the patient on time in case of any disorder. As a consequence, the ECG system has to be optimized in order to ensure a low power consumption for portability, and high computing performance for the real-time acquisition and processing of data. The low power consumption and high performance are usually achieved through highly specialized processors realized as hardwired application specific integrated circuits (ASICs) which can be very efficient, but offer a very-low level of programmability and flexibility. On the other hand, application specific instruction-set processors (ASIPs) are becoming a successful alternative to hardwired circuits thanks to their high performance and efficiency, combined with their programmability. The ECG implementation can much benefit from a highly specialized programmable hardware solution, as offered by ASIPs [1]. ASIPs provide freedom to the ECG for the possible algorithmic changes in the software design. However, to adequately design and configure an ASIP, the knowledge of the application execution characteristics is needed. In other words, the relevant application properties (e.g. execution bottlenecks, type of processing, kind of parallelism etc.) of the application have to be revealed by the analysis tools in order to guide the design and configuration decisions. This paper demonstrates how different application analysis and profiling tools can be exploited for the optimization of the application software, as well as, for the configuration and extension of the corresponding VLIW ASIP processor.

The research reported in this paper was performed in the scope of the European research project ASAM [2] of the ARTEMIS program. ASAM aims at architecture synthesis for ASIPs and ASIP-based systems-on-chip. In [3] the issues and challenges of such architecture synthesis are discussed.

The rest of the paper is organized as follows: Section II briefly explains the ECG system and its heart rate detection algorithm; Section III presents the target processor technology; Section IV focuses on the explanation of the method used for the experiments and quality metrics estimation; Section V discusses the concept of the application analysis and presents the tools used for the analysis, and Section VI presents the software and hardware optimizations with their results. Finally, Section VII concludes the paper and lists some possible future work.

II. Application

This section first introduces the whole ECG system and then explains the part of the ECG system that is realized on the ASIP processor.

A. Electrocardiogram System

An ECG system can be sub-divided into several functional units which monitor the heart electrical activity and process the collected signals. It involves an analog front-end and a digital back-end processing. Figure 1 presents the block diagram of these processing parts, which consists of several units. In the front-end processing block, electrodes, appropriately positioned on the body surface, are used to gather information on the heart electrical activity. Then, the quality of the heart signals is improved by amplification and filtering. In the back-end processing block, the signals are first digitized by analog-to-digital converter (ADC) and subsequently they are used as an input for the *heart rate detection algorithm*.

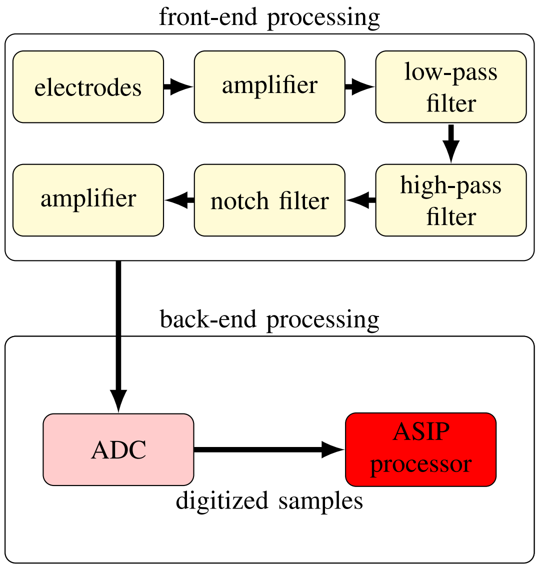

Fig. 1: Electrocardiogram System

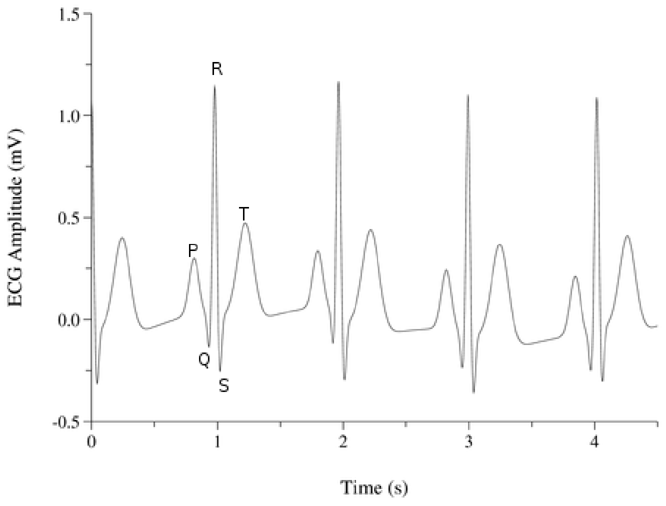

Fig. 2: A waveform of PQRST-complex cycle generated by ECG signal

The electrical signal of the heart consists of a sequence of waves, named P, Q, R, S, T. This sequence constitutes the sinus waveform of the heart signal. Figure 2 shows the aforementioned sequence of waves and the corresponding typical ECG waveform generated by a model as presented in [4]. The temporal sequence of waves remains constant, however the amplitude and the sign of the waves vary depending not only on the heart electrical pulses but also on the position of the electrodes on the body surface. The component corresponding to the main pumping activity of the heart is called QRS, and contains one positive peak R and two negative peaks Q and S. The R-peak constitutes the largest amplitude of the typical waveform. The detection of the R-peak is the most crucial step in the ECG application. The distance between R-peak locations is used to calculate the heart rate.

The ECG application design and synthesis presented in this paper only focuses on the back-end processing of the signals. It does not take into account the front-end processing block, and assumes that signals are successfully retrieved by the front-end and transmitted in the appropriate format to the ASIP-processor through the ADC.

B. Heart Rate Detection Algorithm

The heart rate detection algorithm is the part of the ECG application realized on an ASIP processor. The heart rate is calculated by measuring the time interval that elapses between the two consecutive R-peaks. Consequently, the heart rate detection includes the *R-peak detection algorithm*, which is used for detection of R-peaks in the sinus waveform.

The algorithm uses a buffer of eight elements which is filled by the values sampled through the ADC. The buffer operates in a FIFO mode. The oldest value in the buffer is removed each time a new value is inserted in the buffer. The algorithm analyzes the signal values stored in the buffer by studying the sign of the signal first derivative in the time domain. A peak is detected if the sign of the first derivative changes over the values in the buffer (e.g. it remains positive from sample zero to the sample six and then changes to negative value at position seven). There are two constraints to be checked in order to mark a candidate peak as an actual peak and subsequently calculate the actual heart rate. First of all, a threshold value for the peak amplitude of the current R-peak is computed based on the amplitude of the previous R-peak. In this way, the noise and deflections introduced in the sinus rhythm by the alterations in the patient morphology (for instance when the patient is in motion) are eliminated. If the threshold value constraint is satisfied, a temporary heart rate is calculated by computing the time interval between the current and previous peaks. The second constraint checks if the variation of the heart rate is within a maximum allowable limit in order to compute a definitive heart rate. If this is the case, then the detected R-peak and the corresponding heart rate are marked as the actual values.

The R-peak detection algorithm is kept as simple as possible for the sake of low power consumption and real-time calculation of heart rate requirements. There are several other R-peak detection algorithms such as Martinez et al. [5], Li et al. [6], Hamilton et al. [7], Ghaffari et al. [8], Pan Tompkins et al. [9], Madeiro et al. [10] and Legarreta et al. [11]. The evaluation of the detection quality of the algorithm is not in the scope of our work. It is assumed that algorithm provides acceptable detection quality for a given input data-set.

III. Target Processor Technology

The heart rate detection algorithm is specified in C programming language and compiled on a configurable and extensible ASIP processor. Figure 3 shows a simplified view of the corresponding generic ASIP architecture template.

It includes a VLIW datapath controlled by a sequencer that uses status and control registers, and executes a program

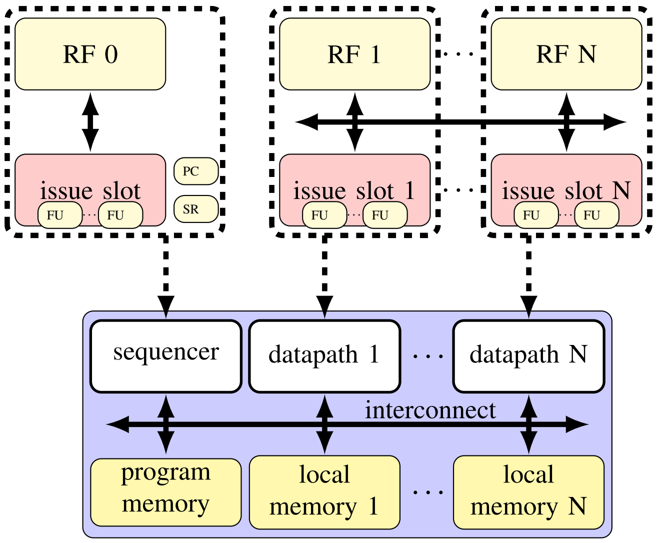

Fig. 3: Generic ASIP architecture template

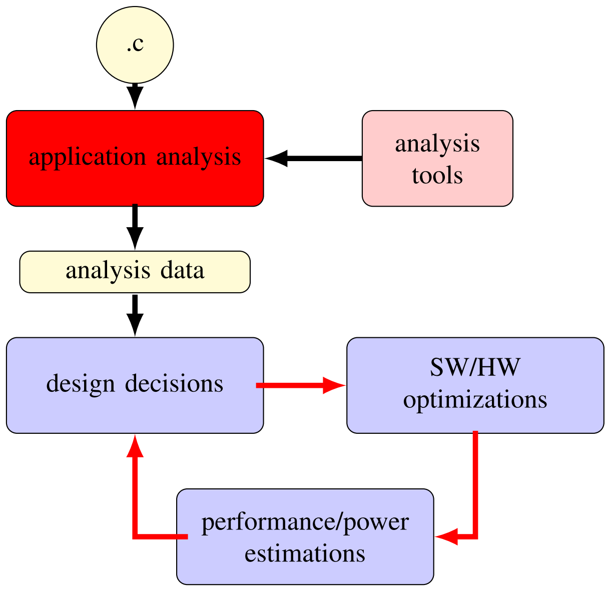

Fig. 4: Application analysis and optimization method

stored in a local program memory. The data path contains functional units organized in several parallel scalar and/or vector issue slots connected via a programmable interconnect network to register files (RF). The functional units perform computation operations on intermediate data stored in the register files. Local memories, collaborating with particular issue slots, enable scalar access for the scalar slots and vector or block access for the vector slots. Both single-instruction multiple-data (SIMD) and multiple-instruction multiple-data (MIMD) processing can be realized on such a processor. The numbers and kinds of functional units, issue slots, register files, memories, interfaces, etc. can freely be selected by specifying component configuration parameters. Moreover, new functional units, issue slots, etc., specific to a particular application, can be developed and added.

IV. Method & Quality Metric Estimation

A. Method

The experiments carried out in the scope of this work were based on the design flow shown in Figure 4. This flow accepts an application C code as an input. In our case, it accepts the C implementation of heart rate detection algorithm from Section II-B. Different analysis tools are exploited in order to extract the relevant data from the application's code. The collected data is used to steer the decisions on software and hardware design by giving information on the achievable improvements provided by a specific optimization. The designer decides which particular optimizations to apply and in which order. The optimizations decided are applied one after the other. The effect of each optimization is evaluated by checking the performance and power consumption metric improvements after each optimization appliance.

B. Performance Estimation

Performance related data are collected through a cycle-accurate simulation of a given algorithm running on a target ASIP processor. The simulation environment analyzes the behavior of the program by providing its number of execution cycles. It also provides information about the instruction-level parallelism (ILP), resource utilization (e.g. issue slot, register, port, function unit etc.) and instructions' schedule.

C. Energy Consumption Estimation

An analytical model is used to estimate the energy consumption of the processor when executing a particular application. The model takes both the active (dynamic) and static energy estimations into account. The model calculates the energy values depending on three parameters: 1) normalized energy and area values for possible hardware components (e.g. function units, register files, logical memories etc.), 2) the current specific configuration of a component (e.g. the number of issue slots and their functional units, the width and capacity of the register files, the number and size of memories and pipeline stages of each function unit etc.), 3) the activity values (utilization) of the each processor component. The utilization indicators are used to estimate active energy and are collected from the simulation environment when running a particular application. These three aforementioned parameters are the arguments of the analytical formula for the active energy calculations for each architecture component. Static energy is the energy being spent in a unit when this unit is not actually used. A unit of static energy is assigned per μm^2. The static energy is estimated based on this assumption.

The script, which carries out the estimation, provides data on the ”energy consumption per application run” in *milli-joule* (mJ). Energy consumption per cycle is calculated from the ”energy consumption per application run” divided by the execution cycle count.

V. Application Analysis

In the following subsections, the method adopted for application analysis is first briefly presented and then the used analysis and profiling tools are introduced.

A. Concept

Application analysis mainly aims to reveal the characteristics of the application. Different (parts of) embedded applications require different kind of processings due to the their various intrinsic characteristics. For instance, the streaming or data-intensive applications (e.g. video or audio coding applications) are more suited to be served by one type of processing, while the control oriented applications (e.g. ECG) are suited to be served by another type of processing. It is therefore crucial to make adequate decisions regarding the kind of processing chosen at the early stages of the ASIP hardware and software synthesis. In order to make these early analysis decisions, it is necessary to characterize an embedded application by analyzing and extracting the different relevant properties of the application code. The characterization corresponds to revealing the relevant properties (e.g. execution bottlenecks, code structure, type of dependencies, data access patterns, kind of parallelism etc.) of the application or application part. This characterization can be performed at different granularities. A decision can be made for the whole application, or for some parts (e.g. kernels, functions, loops) of the application.

The aforementioned concept is not yet realized by automated software tools. In this work, we aim to demonstrate how this concept can be partially performed by making use of the analysis tools.

B. Analysis Tools

This section briefly introduces the analysis and profiling tools exploited to collect data from the application. The first subsection V-B1 explains the tools that are used to guide software optimizations. The second subsection V-B2 presents the tool used to guide hardware optimizations.

1) Tools for software optimization: A profiler, GNU Gprof [12], is used to analyze the program in order to find in which parts of the code the program spent most of its execution time. This execution time distribution provides information on the possible bottlenecks of the application. The profiling data is usually represented as a function call-graph which reveals the execution order of the application code. An example of such call-graph is provided in Figure 5.

GNU Gcov [13] is used to test the static code coverage, i.e. how much of the program is executed under the stimulus of input data. In other words, it checks the quality in terms of representativity of the considered set of the input data. Experiment on ECG shows that the used input data stimulate the execution of 92.5 % of the program. Furthermore, analysis passes of the LLVM (Low Level Virtual Machine) compiler [14] are used to extract relevant information from the application such as code structure. An example of extracted code structure is given in Figure 6.

2) Tools for hardware optimization: μProfiler [15], [16] is used to guide the hardware optimizations. μProfiler provides information such as the execution frequencies of different arithmetic and logical operations, frequently used C data types together with their value ranges. The obtained profiling data can drive the decisions on the target architecture such as removing or adding functional units of the ASIP, modifying bit-width of instructions and fixing registers size, etc. Moreover, it also aims to aid designers to take decisions about the memory structure (e.g. number and size of memories) by providing profiling information such as the total amount of each kind of memory considered (e.g. static, run-time stack), the most heavily accessed data, the most memory intensive portions of the application. There are other several tools such as SpecC Profiler [17] and Software Instrumentation Tool (SIT) [18] that can be potentially used to guide hardware optimizations.

VI. Design Optimizations

A. Experimental Setup

The heart rate detection algorithm is initialy compiled on two existing 32-bit VLIW ASIP processors. Among these two possible processors, the base processor is selected to start the design optimizations. Initial estimations of performance and energy consumption steer the selection of such a base processor. For these estimations, no optimizations are applied to the software or hardware specifications. Table I shows the performance in terms of execution cycle count and energy consumption when the original code is mapped on the two different scalar VLIW processors: a) the *tad* processor which has one sequencer issue slot, and b) the *pearl* processor which has one sequencer and one datapath issue slot.

TABLE I: Initial quality metrics estimation for the original code

	Tad (1-issue slot)	Pearl (2-issue slots)
Cycle Count	372389	277464
Energy (mJ)	0.2224	0.3459
Energy (mJ/cycle)	0.597	1.247
ILP	0.88	1.31

Since the *pearl* processor is more complex (it has two issue-slots) than the *tad* processor, the compiler has more opportunity to exploit instruction level parallelism (ILP). Therefore, the total cycle count for *pearl* is lower than for *tad*. On the other hand, the energy consumption for *pearl* is higher than that estimated for *tad* due to the high number of utilized resources. Moreover, each *pearl* additional resource introduces additional static energy consumption. However, starting with a more complex processor avoids initial hardware resource limitations that could prevent parallelism opportunities exploitation. Consequently, the *pearl* processor is selected as a base processor.

B. Software Optimizations

This section explains the software optimizations applied to the heart rate detection algorithm which is mapped onto the base processor. Software optimizations refer to modifications of application code only; therefore, in a first moment no changes are applied to the base processor.

The compute-intensive parts (bottlenecks) of the application is the first target for software optimizations. This is due to the fact that a bottleneck optimization result in a significant improvement for the whole implementation. Figure 5 shows the profiled call-graph of the heart rate detection algorithm. The call-graph provides information on the relative execution time distribution of the functions. The data is collected during the execution of the heart rate detection algorithm on general purpose processor with 1653 digitized input samples.

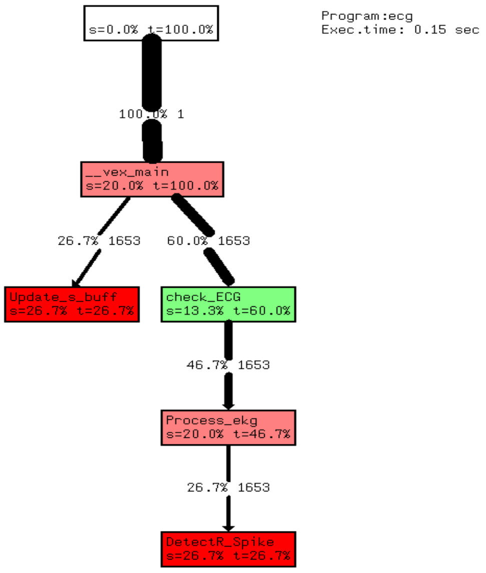

Fig. 5: Profiled call-graph of heart rate detection algorithm (obtained using GNU Gprof)

The execution time distribution guides bottlenecks identification to start the software optimizations. Therefore, the functions *Update_s_buff*, *DetectR_Spike* and *Process_ekg* are the first three candidates for optimization. The first function updates the buffer, the second function detects the R-peak and finally the third function checks constraints to decide actual peak and heart rate. These three functions constitute approximately 74% (26.7% + 26.7% + 20%) of the total execution time.

Subsequently, the code structure of the each kernel is analyzed in order to steer the selection of the potentially beneficial optimization type for that kind of structure. Figure 6 shows the code structure of the function *Update_s_buff* as an example. The code structure reveals the type of statements and their structure in the code. The analysis of the three bottleneck functions shows that the heart rate detection algorithm includes substantial amount of control statements (e.g. if-then-else) which are preponderant with respect to the data oriented statements, i.e. all the statements involved in a loop nest (e.g. do-while, for). Therefore, the optimizations are mainly targetted to improve the control structures in the code.

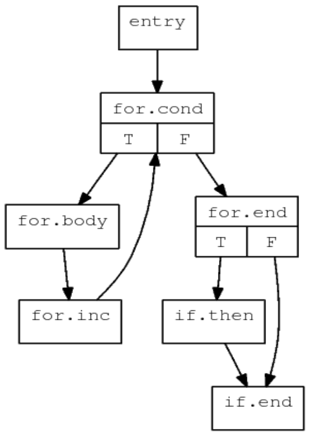

Fig. 6: Code structure of the update buffer function (obtained using LLVM)

Experiments show that following four code optimizations are the most beneficial.

1) Function Inlining: It is applied to the functions in the code and replaces the function calls with the body of the corresponding functions. This optimization eliminates jumps caused by function call and return statements. As a consequence, it provides more possibilities to the ASIP scheduler to better exploit the ILP.

2) Loop Unrolling: Loops introduce overhead due to the branch conditionals that control the flow of the loops. This overhead can be minimized by reducing the number of iterations and replicating the loop body. As a result, a larger loop body is produced and a larger set of instructions can be scheduled by the ASIP scheduler. This enables the possibility to better exploit parallel functional units and pipelining mechanisms. In the case of the ECG application, the loop iterate only over eight samples to detect a peak, thus a full unrolling of the loop nest can represent a reasonable optimization choice.

3) If Conversion: It is applied in order to transform *if-then-else* constructions into multiplexer-like operations. The classical if-then-else constructs are written in a C ternary conditional operator format. Then, they are interpreted by the compiler as multiplexer operations. Moreover, if-conversion is used to remove if-statements inside the loop bodies in order to allow loop unrolling.

4) Control Flow Optimizations: In addition to the if conversion, another technique is also used to eliminate control structure. The number of boolean variable test statements are reduced by replacing them with boolean decision variables. The value of such a variable is updated during the the program execution and, finally, the jump is carried out at the end of the execution depending on the current value of the boolean variable.

The common property of these four optimizations is the fact that all of them increase the length of straight line code blocks which gives the scheduler more opportunity to exploit ILP. Table II shows the effect of these software optimizations with the cumulative performance improvement.

TABLE II: Software Optimizations

Optimization Type	Cycle Count	Speed-up
Original	277464	-
Function Inlining	178318	35 %
Loop Unrolling	143569	48 %
If-conversion	133644	52 %
Control flow opt.	119347	57 %

Table III compares the cycle count, energy consumption and energy consumption per cycle values of the original and software optimized code. The software optimizations result approximately in 57% speed-up for the whole application run on *pearl* processor. The energy consumption is also reduced by 30% due to the reduced cyle count. On the other hand, energy consumption per cycle is increased due to the fact that improvement in speed-up is much more higher than the improvement in energy consumption.

TABLE III: Results after the software optimizations

	Cycle Count	Energy (mj)	Energy (mj/cycle)
Original	277464	0.3459	1.247
Software optimized	119347	0.2491	2.087

C. Hardware Optimizations

This section explains the hardware optimizations applied to the base processor. The experiments presented in this section are carried out after software optimizations, and take the base processor and the optimized software as a starting point. Hardware optimizations that modify the processor depend on the analysis results. The analysis results are gathered by profiling the application with μProfiler. The hardware optimizations include:

1) Data-width Reduction: Tables IV, V, VI present the some results of application analysis. Table IV presents the used data types and the percentage of their usage in the application. It shows that computations are only performed on integer data type (e.g. *int*, *uchar*, *ushort*). Memory read and write operations are carried out through *pointers* which also have integer types. *Long* and *floating-point* (e.g. *float*, *double*) data types are not used in the application. Therefore, the final processor does not need to have function unit that supports floating-point and long operations.

TABLE IV: Data types and their usage

int (%)	uchar (%)	ushort (%)	pointer (%)
40.07	10.74	34.11	15.08

Table V presents the approximated distribution of the data types usage with respect to different operation types. It shows that arithmetic/logic and multiplication operations are used with *int*, *uchar*, *ushort* and *pointer* types. Memory operations are performed on *uchar* and *ushort* types. Moreover, shift and relational operations are carried out only on integer data type.

TABLE V: Data type usage w.r.t operation type

	Arith/Logic	Mult.	Memory	Shift	Relational
int (%)	47	54	0	100	100
uchar (%)	10	26	92	0	0
ushort (%)	8	20	8	0	0
pointer (%)	35	0	0	0	0

Table VI demonstrates the value ranges of data types. It shows the maximum and minimum values assigned to the particular data type. In the last row also the maximum and minimum values of the immediates (i.e. constants) used in the application code are shown.

TABLE VI: Value ranges of data types

	Max. Value	Min. Value
int	30000	0
unsigned short	4095	0
unsigned char	8	0
immediates	4095	0

Correlated analysis of data collected in tables IV, V and VI shows that immediates and input samples of the heart rate detection algorithm can be stored and processed by using a 16-bit architecture. Therefore, the data-width of the base processor is reduced from 32-bit to 16-bit. Thanks to the this optimization and as shown in Table VIII, the power consumption is reduced approximately by half. Moreover, there is also an improvement in the application performance. This is due to the fact that data-width reduction eliminates some extra operations performed during the load of immediates (e.g. shifting the 16-bit data in 32-bit register in order to align data for computation).

2) Communication Optimization: Manual analysis of the communication structure of the processor allows to detect potential bottlenecks in communication structure that may prevent possible performance gains. To this purpose, the ports of the two register files are increased from 1x2 (number of input ports x number of output ports) to 2x3. Table VIII shows that communication optimizations introduce a significant improvement in performance without any overhead in energy consumption. This is due to the fact that increase in static energy consumption is compensated by the reduction in dynamic energy consumption.

3) Resource Minimization: Resource minimization aims to minimize the number of resources either by resizing them or by completely removing the useless or under-utilized resources from the processor specification. Figure 7 shows the various operation types that are used in the application with their usage percentage. Arithmetic corresponds to addition type operations (+,-), memory corresponds to load, store operations, and relational corresponds to operations such as ($<$, $>$, $\leq$, $\geq$, $==$). Only negation (!) is used as an unary operation and left shift ($<<$) is the only used shift operator. This analysis guides the resource minimization with respect to the function unit and their corresponding operations. Moreover, other resources such as size of the registers, number of register in the register files and size of memories are subject to resouce minimization. These optimizations are performed by considering the range of used data values presented in Table VI, as well as, the life-time of intermediary variables which influence the size of register files.

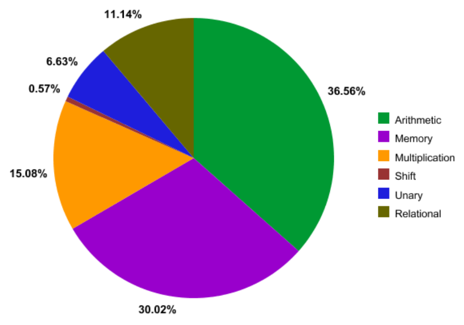

Fig. 7: Operators usage statistics

4) Instruction-width Optimization: A VLIW instruction includes bits to select registers, operations, busses and bits to encode immediate values. Removing function units and their corresponding operations, reducing the number of registers in register files and decreasing the number of bits required to encode immediate values brings to a reduced size of VLIW instruction word which is set to 75 bits. The related entry in the Table VIII presents the combined improvements due to the resource minimization and the instruction-width reduction. It shows that there is a reduction in energy consumption mainly due to the shrinked program memory (as a result of instruction word reduction). On the other hand, an overhead is introduced in performance. This is because of the resouce minimization results in decreased ILP.

TABLE VII: Variable access statistics

Variables	Access (%)
s_buff	63.42
sample	12.87
sample_time	7.71
is_picco	5.14
min_val	3.0
others	7.86

5) Data Distribution: Table VII shows the frequently accessed variables during the execution of the application. This analysis guides the decision on the number of required local memories. Moreover, it also allows to distribute the heavily accessed data over instantiated memories in order to allow parallel data access. However, the heart rate detection algorithm processes data contained only in one buffer (*s_buff*, 63.42 %). In this case, it makes sense to keep current design which includes one data memory. This is true because of the data dependencies and also the algorithm is not data-dominated but control oriented. Thus, data distribution on parallel resources with the corresponding mechanism to manage data access may cause an increase of computation complexity more than a temporal performance improvement.

Table VIII summarizes the hardware optimizations and their results in terms of energy consumption per application run and total number of cycle count. As a consequence of software and hardware optimization, 62% improvement is achieved for speed-up and energy consumption. Furthermore, total logic and memory area is decreased by 58% mainly due to the resource minimization, data-width and instruction-width reduction.

TABLE VIII: Results of the hardware optimizations

Optimization Type	Cycle Count	Energy (mj)
Software Optimized	119347	0.2491
Data-width Reduction	116111	0.1343
Communication Opt.	99639	0.1306
Resource Min.& Ins-width Opt.	102780	0.1299

VII. Conclusion and Future Work

In this paper, we present the analysis and optimization of the ECG application. Our design flow is based on two main groups of analysis tools, aimed at software and hardware optimizations, respectively. The software optimizations exploit data analysis information on the program execution time and the code structure, which point out the bottlenecks of the application and the most beneficial software optimizations for the ECG. Such optimizations are mostly aimed at improving control-oriented code. The hardware optimizations exploit data collected by μProfiler to reduce data and instruction width, minimize useless and under-utilized resources, and optimize the system communication.

Both software and hardware optimizations are tested against improvement metrics such as performance and energy consumption. The software optimization only bring to an ECG application speed-up of 57% and an "energy consumption per application run" reduction of 30%, although the energy per cycle is increased due to the optimizations themselves. The cumulative speed-up due to the appliance of both software and hardware optimizations is up to 62% and the reduction of energy consumption per application run is also up to 62%. As a consequence of performed experiments, we have observed a direct link between application analysis and the steering of possible software and hardware improvements, for the targetted ASIP technology. Future works are aimed at formalization and automation of the proposed concept.

Acknowledgment

The authors of this paper are indebted to all partners of the European research project ASAM of the ARTEMIS Program.

References

[1] Y. Yassin, P. Kjeldsberg, J. Hulzink, I. Romero, and J. Huisken, "Ultra low power application specific instruction-set processor design for a cardiac beat detector algorithm," 2009.

[2] ASAM project website. [Online]. Available: http://www.asam-project.org/

[3] L. Jozwiak and M. Lindwer, "Issues and challenges in development of massively-parallel heterogeneous MPSoCs based on adaptable ASIPs," in *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 483–487. [Online]. Available: http://dx.doi.org/10.1109/PDP.2011.55

[4] P. E. Mcsharry, G. D. Clifford, L. Tarassenko, and L. A. Smith, "A dynamical model for generating synthetic electrocardiogram signals," *IEEE Transactions on Biomedical Engineering*, vol. 50, pp. 289–294, 2003.

[5] J. P. Martnez, R. Almeida, S. Olmos, A. P. Rocha, and P. Laguna, "A wavelet-based ECG delineator: evaluation on standard databases." *IEEE Transactions on Biomedical Engineering*, vol. 51, no. 4, pp. 570–581, 2004. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/15072211

[6] C. Li, C. Zheng, and C. Tai, "Detection of ECG characteristic points using wavelet transforms," *IEEE Transactions on Biomedical Engineering*, vol. 42, no. 1, pp. 21–28, 1995. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=362922

[7] P. S. Hamilton and W. J. Tompkins, "Quantitative investigation of QRS detection rules using the MIT/BIH arrhythmia database." *IEEE Transactions on Biomedical Engineering*, vol. 33, no. 12, pp. 1157–1165, 1986. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/3817849

[8] A. Ghaffari, H. Golbayani, and M. Ghasemi, "A new mathematical based QRS detector using continuous wavelet transform," *Comput. Electr. Eng.*, vol. 34, pp. 81–91, March 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1347462.1347734

[9] J. Pan and W. J. Tompkins, "A real-time QRS detection algorithm." *IEEE Transactions on Biomedical Engineering*, vol. 32, no. 3, pp. 230–236, 1985. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/3997178

[10] J. P. V. Madeiro, P. C. Cortez, F. I. Oliveira, and R. S. Siqueira, "A new approach to QRS segmentation based on wavelet bases and adaptive threshold technique." *Medical Engineering Physics*, vol. 29, no. 1, pp. 26–37, 2007. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/16500133

[11] I. Romero Legarreta, P. Addison, M. Reed, N. Grubb, G. Clegg, C. Robertson, and J. Watson, "Continuous wavelet transform modulus maxima analysis of the electrocardiogram: beat characterisation and beat-to-beat measurement." *Int. J. Wavelets Multiresolut. Inf. Process.*, vol. 3, no. 1, pp. 19–42, 2005.

[12] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *SIGPLAN Not.*, vol. 39, pp. 49–57, April 2004. [Online]. Available: http://doi.acm.org/10.1145/989393.989401

[13] Gnu gcov website. [Online]. Available: http://gcc.gnu.org

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[15] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained application source code profiling for asip design," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 329–334. [Online]. Available: http://doi.acm.org/10.1145/1065579.1065666

[16] K. Karuri, C. Huben, R. Leupers, G. Ascheid, and H. Meyr, "Memory access micro-profiling for asip design," in *Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 255–262. [Online]. Available: http://dl.acm.org/citation.cfm?id=1109224.1109389

[17] L. Cai, A. Gerstlauer, and D. Gajski, "Retargetable profiling for rapid, early system-level design space exploration," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 281–286. [Online]. Available: http://doi.acm.org/10.1145/996566.996651

[18] M. Ravasi and M. Mattavelli, "High-level algorithmic complexity evaluation for system design," *J. Syst. Archit.*, vol. 48, pp. 403–427, May 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=859252.859256